Politecnico di Torino

Collegio di Elettronica, Telecomunicazioni e Fisica

Laboratory 3 : Design of a RISC-V-lite processor

Report for the course
Integrated Systems Architecture

Master degree in Electronics Engineering

Authors: Group 09

R. Capruzzi(300142), R. Giunti(301167), O. Laouibi(303331)

January 16, 2024

Contents

CHAPTER 1

Part 1: Compiling the source application

1.1 Minv application

1.1.1 Description

The goal of this activity has been to design in VHDL a RISC-V-lite processor architecture with 5 pipeline stages. The instructions that the implemented processor supports are a subset of the whole RV32I, which has a 32-bit fixed-width instruction set, where instructions are organized in six classes: R, I, S, SB, U and UJ. In particular, this subset is defined by the instructions required to execute the *minv* application given as specification, which we will talk about now.

1.1.2 Compiling the minv application

The application to run in the architecture is made of 3 files written in C language, in particular they are:

- a file named *main.c* that defines a vector of length N which is then passed to the minv function, which gives results that are then printed;
- the *minv.h* header file that is called upon the call of the function;
- the *minv.c* file containing the description of the function, which performs the absolute value of every element of the vector and finds the largest one.

The codes are here reported, in order:

main.c :

```
//#define IS86

#ifdef IS86
#include <stdio.h>
#endif
#include "minv.h"

#define N 7

int v[N] = {9, −46, 21, −2, 14, 26, −3};
volatile int m;

int main()
```

```
{

  m = minv(v, N);

#ifdef IS86
  printf("%d\n", m);
#endif
  return 0;
}
```

minv.h :

```
#ifndef MINV_H
#define MINV_H

int minv(int *v, int N);

#endif
```

minv.c :

```
#include"minv.h"

#define myabs(x) (((x) < 0) ? (-(x)) : (x))

int minv(int *v, int N)
{
  int i;
  int m;

  m = myabs(v[0]);
  for (i=1; i<N; i++)
  {
    if (m > myabs(v[i]))
      m = myabs(v[i]);
  }

  return m;
}
```

The goal with this application is to obtain the sequence of the 32 bits instructions to execute. With a list of all the instructions making up the subset of the RV32I, the architecture can then be designed to support all of them, avoiding the area overhead of a processor capable of executing instructions not required by our needs.

The process to obtain the wanted instructions is now described. The first step is to rely on *gcc* to generate the executable file from the *main* application; to do this, the compiler requires information about the memory organization and a preamble to initialize registers. The registers initialization is performed by the *crt0.s* file, which was provided to us and is now reported:

```
.section .init, "ax"
.global _start
_start:
    .cfi_startproc
    .cfi_undefined ra
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop
    la sp, __stack_top
    add s0, sp, zero
    jal ra, main
el:
    j el
    .cfi_endproc
    .end
```

This code defines the **start** symbol, which represents the address of the first instruction to execute, then it loads the global pointer and stack pointer registers (gp and sp) and finally calls the main function with a jump-and-link instruction (jal). The execution terminates with an infinite loop (j el).

The information about the memory organization is performed by the linked script: a file named *riscV32-isa.ld* which sets the first address at *0x00400000* for the instruction ROM memory, and *0x10010000* for the data RAM. It sets also the global pointer to *0x100008000*, the stack pointer to *0x7fffeffc* and assigns each segment of the program to the correct memory type. The code for this file is here reported:

```
OUTPUT_FORMAT("elf32-littleriscv", "elf32-littleriscv",
              "elf32-littleriscv")
OUTPUT_ARCH(riscv)

MEMORY
{
   /* ISA lab begin */
   ROM (rx)    : ORIGIN = 0x00400000, LENGTH = 64k
   RAM (rwx)   : ORIGIN = 0x10010000, LENGTH = 4000M
   /* ISA lab end */
}

ENTRY(_start)

SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x10000)); . = SEGMENT_START("text-segment",
  /* ISA lab begin */
  PROVIDE(__stack_top = 0x7fffeffc);
  /* ISA lab end */
  .init          :
  {
    KEEP (*(SORT_NONE(.init)))
  } > ROM
  .text          :
  {
    *(.text.unlikely .text.*_unlikely .text.unlikely.*)
    *(.text.exit .text.exit.*)
    *(.text.startup .text.startup.*)
    *(.text.hot .text.hot.*)
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
  } > ROM

  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
  .data          :
  {
    *(.data .data.* .gnu.linkonce.d.*)
    SORT(CONSTRUCTORS)
  } > RAM
  .data1          : { *(.data1) } > RAM
  .got            : { *(.got.plt) *(.igot.plt) *(.got) *(.igot) } > RAM
  /* We want the small data sections together, so single-instruction offsets
     can access them all, and initialized data all before uninitialized, so
     we can shorten the on-disk segment size.  */
  .sdata          :
  {
   /* __global_pointer$ = . + 0x800; */
   /* ISA lab begin */
   __global_pointer$ = 0x10008000;
   /* ISA lab end */
    *(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata.cst2) *(.srodata .srodata.*)
    *(.sdata .sdata.* .gnu.linkonce.s.*)
  } > RAM
   _edata = .; PROVIDE (edata = .);
  . = .;
    __bss_start = .;
  .sbss           :
  {
    *(.dynsbss)
    *(.sbss .sbss.* .gnu.linkonce.sb.*)
    *(.scommon)
  } > RAM
  .bss            :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.
      FIXME: Why do we need it? When there is no .bss section, we don't
      pad the .data section.  */
   . = ALIGN(. != 0 ? 32 / 8 : 1);
  } > RAM
  . = ALIGN(32 / 8);
  . = SEGMENT_START("ldata-segment", .);
  . = ALIGN(32 / 8);
   _end = .; PROVIDE (end = .);
}
```

After generating the executable with the RISC-V version of *gcc*, that is *riscv32-unknown-elf-gcc*, we relied on the disassembler (*objdump*) to convert the executable file into a text file. In particular, from the executable file it was necessary to extract both the sequence of instructions with the corresponding address and the data. The obtained files from this procedure are the *main.hex* and *data.hex*, which are now reported:

main.hex :

```
main:     file format elf32-littleriscv


Disassembly of section .init:

00400000 <_start>:
  400000:       1fc18197                auipc   gp,0x1fc18
```

```
//add upper immediate to PC
  400004:        01c18193                addi    gp,gp,28 # 2001801c <__global_pointer$> //add immediate
  400008:        7fbff117                auipc   sp,0x7fbff
  40000c:        ff410113                addi    sp,sp,-12 # 7fffeffc <__stack_top>
  400010:        00010433                add     s0,sp,zero
  400014:        008000ef                jal     ra,40001c <main>

00400018 <el>:
  400018:        0000006f                j       400018 <el>

Disassembly of section .text:

0040001c <main>:
  40001c:        10010537                lui     a0,0x10010
  400020:        ff010113                addi    sp,sp,-16
  400024:        00700593                li      a1,7
  400028:        00050513                mv      a0,a0
  40002c:        00112623                sw      ra,12(sp)
  400030:        01c000ef                jal     ra,40004c <minv>
  400034:        00c12083                lw      ra,12(sp)
  400038:        100107b7                lui     a5,0x10010
  40003c:        00a7ae23                sw      a0,28(a5) # 1001001c <_edata>
  400040:        00000513                li      a0,0
  400044:        01010113                addi    sp,sp,16
  400048:        00008067                ret

0040004c <minv>:
  40004c:        00052603                lw      a2,0(a0) # 10010000 <v>
  400050:        00100713                li      a4,1
  400054:        41f65793                srai    a5,a2,0x1f
  400058:        00c7c633                xor     a2,a5,a2
  40005c:        40f60633                sub     a2,a2,a5
  400060:        02b75863                ble     a1,a4,400090 <minv+0x44>
  400064:        00259593                slli    a1,a1,0x2
  400068:        00450713                addi    a4,a0,4
  40006c:        00b50533                add     a0,a0,a1
  400070:        00072783                lw      a5,0(a4)
  400074:        00470713                addi    a4,a4,4
  400078:        41f7d693                srai    a3,a5,0x1f
  40007c:        00f6c7b3                xor     a5,a3,a5
  400080:        40d787b3                sub     a5,a5,a3
  400084:        00c7d463                ble     a2,a5,40008c <minv+0x40>
  400088:        00078613                mv      a2,a5
  40008c:        fee512e3                bne     a0,a4,400070 <minv+0x24>
  400090:        00060513                mv      a0,a2
  400094:        00008067                ret
```

data.hex :

```
main:     file format elf32-littleriscv


Disassembly of section .data:

10010000 <v>:
10010000:        0009                    c.addi  zero,2
10010002:        0000                    unimp
10010004:        ffd2                    fsw     fs4,252(sp)
10010006:        ffff                    0xffff
10010008:        0015                    c.addi  zero,5
1001000a:        0000                    unimp
```

```
1001000c:       fffe            fsw     ft11,252(sp)
1001000e:       ffff            0xffff
10010010:       000e            0xe
10010012:       0000            unimp
10010014:       001a            0x1a
10010016:       0000            unimp
10010018:       fffd            bnez    a5,10010016 <v+0x16>
1001001a:       ffff            0xffff
```

As it can be seen, the *main* file is composed of three columns of code, representing the memory address, the instruction in hexadecimal form and the corresponding assembly instruction, while the *data* file contains the data stored in the memory and the address where they are stored. Other than that, the instructions presented in this file have to be discarded, since they are a result of the disassembler trying to translate the data in to instructions, which in result have no meaning and are not relevant.

The final goal with the data and instructions obtained by compiling the application is to generate an assembly file ready to be simulated with the RISC-V simulator named *RARS*. To do this 2 codes (named *extract_code.sh* and *extract_data.sh*) have been utilized to extract the binary executable in hexadecimal format from the *main.hex* file and the data from the *data.hex* one. The resulting files are the *main.asm* and the *data.txt*, which are now reported:

main.asm :

```
section .text:

__start:
auipc gp,0x1fc18
addi gp,gp,28
auipc sp,0x7fbff
addi sp,sp,-12
add s0,sp,zero
jal ra,40001c <main>

el:
j 400018 <el>



main:
lui a0,0x10010
addi sp,sp,-16
li a1,7
mv a0,a0
sw ra,12(sp)
jal ra,40004c <minv>
lw ra,12(sp)
lui a5,0x10010
sw a0,28(a5)
li a0,0
addi sp,sp,16
ret

minv:
lw a2,0(a0)
li a4,1
srai a5,a2,0x1f
xor a2,a5,a2
sub a2,a2,a5
ble a1,a4,400090 <minv+0x44>
```

```
slli a1,a1,0x2
addi a4,a0,4
add a0,a0,a1
lw a5,0(a4)
addi a4,a4,4
srai a3,a5,0x1f
xor a5,a3,a5
sub a5,a5,a3
ble a2,a5,40008c <minv+0x40>
mv a2,a5
bne a0,a4,400070 <minv+0x24>
mv a0,a2
ret
```

data.txt :

```
v:
0009
0000
ffd2
ffff
0015
0000
fffe
ffff
000e
0000
001a
0000
fffd
ffff
```

From these last two files, the final *program.asm* file, which was used for the simulation in *RARS*, has been obtained by taking these two files and apply some minor editing. The resulting file is now presented:

```
                .data
v:
                .word 0x00000009
                .word 0xffffffd2
                .word 0x00000015
                .word 0xfffffffe
                .word 0x0000000e
                .word 0x0000001a
                .word 0xfffffffd

                .text
                .globl __start

__start:
                auipc gp,0x1fc18
                addi gp,gp,28
                auipc sp,0x7fbff
                addi sp,sp,-12
                add s0,sp,zero
                jal ra,main

el:
                j el
```

```
main:
                    lui a0,0x10010
                    addi sp,sp,-16
                    li a1,7
                    mv a0,a0
                    sw ra,12(sp)
                    jal ra,minv
                    lw ra,12(sp)
                    lui a5,0x10010
                    sw a0,28(a5)
                    li a0,0
                    addi sp,sp,16
                    ret

minv:
                    lw a2,0(a0)
                    li a4,1
                    srai a5,a2,0x1f
                    xor a2,a5,a2
                    sub a2,a2,a5
                    ble a1,a4,less_or_eq1
                    slli a1,a1,0x2
                    addi a4,a0,4
                    add a0,a0,a1

branch_neq:
                    lw a5,0(a4)
                    addi a4,a4,4
                    srai a3,a5,0x1f
                    xor a5,a3,a5
                    sub a5,a5,a3
                    ble a2,a5,less_or_eq2
                    mv a2,a5

less_or_eq2:
                    bne a0,a4,branch_neq

less_or_eq1:
                    mv a0,a2
                    ret
```

The program *RARS* simulates the execution of these instructions, updating the register file and data memory at every iteration, representing a reference for how the structures in the RISC-V architecture have to behave for every instruction.

CHAPTER 2

Design of the architecture

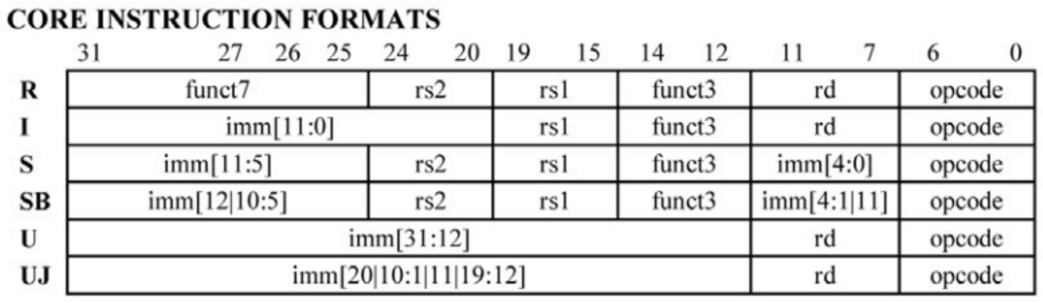
2.1 Instructions supported

The first step in designing the architecture has been to understand which instructions it had to support, in this way the data path can be designed in the most efficient way, avoiding the area overhead of components not required by the application. The complete list of instructions that can be derived from the application is now reported, including a brief description of what each instruction does and which format they are part of.

	Instruction	Description	Format	Opcode/funct3/funct7
add	R[rd], R[rs1], R[rs2]	R[rd] = R[rs1] + R[rs2]	R	0110011/000/0000000
sub	R[rd], R[rs1], R[rs2]	R[rd] = R[rs1] - R[rs2]	R	0110011/000/0100000
xor	R[rd], R[rs1], R[rs2]	R[rd] = R[rs1] $\oplus$ R[rs2]	R	0110011/100/0000000
addi	R[rd], R[rs1], Imm	R[rd] = R[rs1] + Imm	I	0010011/000
lw	R[rd], Imm(R[rs1])	R[rd] $\leftarrow$ M[R[rs1] + Imm]	I	0010011/010
srai	R[rd], R[rs1], Imm	R[rd] = R[rs1] $\gg$ Imm	I	0010011/101
slli	R[rd], R[rs1], Imm	R[rd] = R[rs1] $\ll$ Imm	I	0010011/001
jalr	R[rd], Imm(R[rs1])	R[rd] = PC+4; PC = R[rs1]+Imm	I	1100111/000
sw	R[rs2], Imm(R[rs1])	M[R[rs1]+Imm] $\leftarrow$ R[rs2]	S	0100011/010
bne	R[rs1], R[rs2], Imm	if(R[rs2] $\neq$ R[rs1]) $\Rightarrow$ PC=PC+Imm	SB	1100011/001
bge	R[rs1], R[rs2], Imm	if(R[rs1] $\geq$ R[rs2]) $\Rightarrow$ PC=PC+Imm	SB	1100011/101
auipc	R[rd], Imm	R[rd] = PC + Imm, 12'b0	U	0010111
lui	R[rd], Imm	R[rd] = 32b'Imm, 12b'0	U	0110111
jal	R[rd], Imm	R[rd] = PC+4; PC=PC+Imm	UJ	1101111

Table 2.1: List of instructions required by the application

The 32-bit description of each format was taken directly from the RISC-V reference data, and it is the following:



CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Figure 2.1: Description of each format in the RISC-V architecture

As can be seen from Table 2.1 and Figure 2.1 each format is defined by:

- an opcode, made of 7 bits;
- 2 additional function fields, made of 3 and 7 bits, that define the different type of operations in the different formats;
- the rs1, rs2 and rd fields, made of 5 bits each, that define respectively the 2 source registers and the destination register from which to take and return the operands needed to perform the operation.

2.2 Design description

The goal of the design was being able to run the code of the *minv* application exactly as it is, without changing any instruction, and maximizing throughput. The design started with the implementation of the non-pipelined version, to test the correctness of the architecture, and was later augmented with pipeline stages, which are 5 in the RISC-V architecture. Other than this, the Harvard architecture was requested as specification, meaning that there must be separate physical memory for instructions and data, assuming that no assumptions have to be made on the number of clock cycles required to read and write a value from a memory, i.e. the architecture can work with any memory technology and does not rely on a specific one. A description of the design will now be given, starting from the easiest non pipelined version until the final pipelined one with all the elements and considerations done to maximize throughput.

2.2.1 Non-pipelined RISC-V

The complete design of the non pipelined RISC-V can be seen in the next figure, along the control unit that generates all the control signals required to make the architecture handle instructions from different formats. This design is a theoretical one, since it assumes every element of the design is combinatorial, except the program counter which fetches one instruction every clock cycle, but since the memories making up the instruction memory and the data memory, as well as the register file, are sequential rather than combinatorial, the timing of all elements is not respected and thus the circuit cannot function. Nonetheless it's easier to describe the design starting from this implementation.

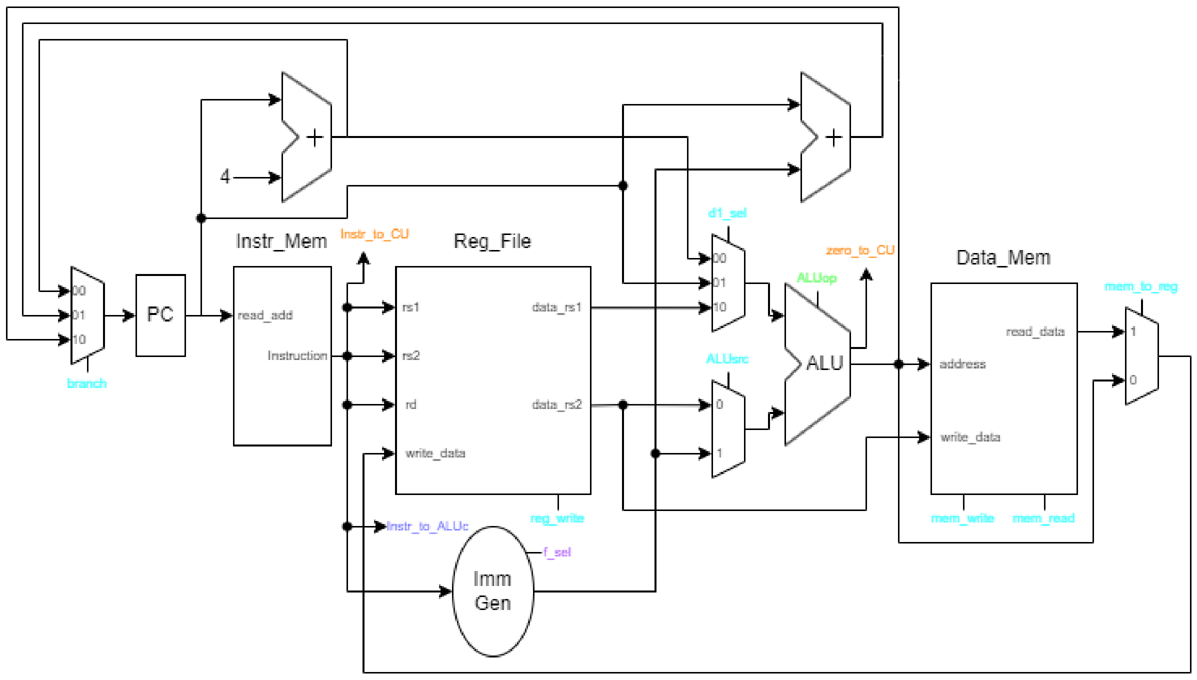


Figure 2.2: Non pipelined architecture of the RISC-V

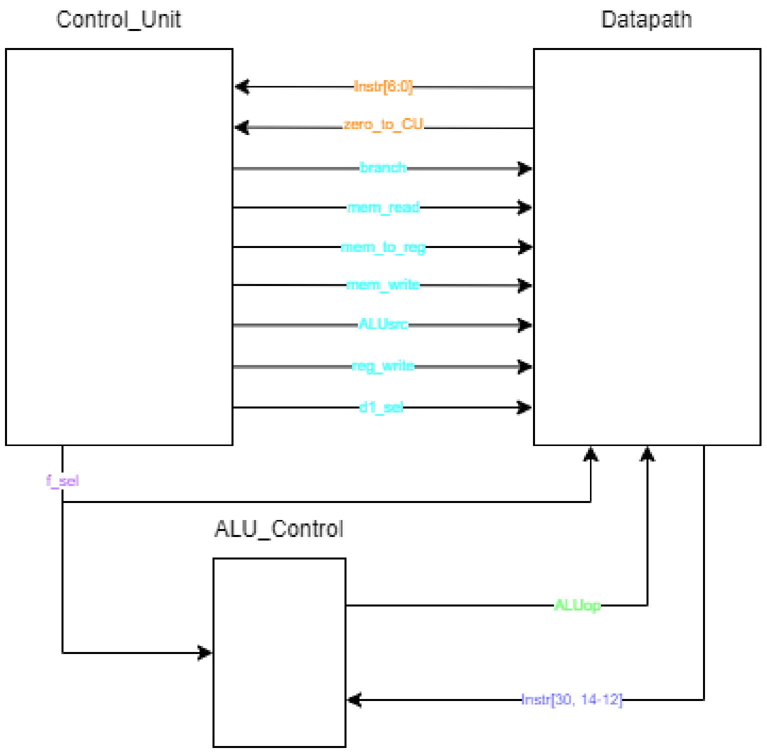


Figure 2.3: Control Unit and ALU Control of the RISC-V

Here are several details about some elements of the design.
The program counter (PC) is a simple register of 32-bits, with a starting value equal to *0x00400000*,

corresponding to the first address of the instruction memory. The instruction memory, as well as the data memory, have been implemented utilizing the provided *sram_32_1024_freepdk45*, and their memory content is initialized by a command written in the test bench. In the case of the instruction memory, the memory is filled with the 32-bits instructions of the *minv* application to execute, which are 38 (the other addresses of the RAM are filled with zeroes).

The register file is an array of 32 standard logic vectors of 32 bits (a memory of 32 x 32), which has 3 inputs of 5 bits representing the address for the registers containing the 2 source operands (rs1 and rs2) and the destination register (rd), one input of 32 bits representing the data to store in the destination register, and two outputs representing the content of the registers indexed by the address rs1 and rs2.

The ALU performs the operations required by our application, which are the sum and subtraction of signed numbers, the logic AND, OR and XOR between the two input operands, and the right shift and left shift operation of the first operand by an amount dictated by the second operand.

The ALU control tells the ALU which operation to perform by taking two bits from the control unit as well as the function 3 and one bit of the function 7 (see Table 2.1).

The Control Unit is what, based on the opcode and function 3 bits, is able to decipher which instruction to execute and generates the control signals that make all the different hardware components perform the right operation (such as the register write signal of the register file, or the memory read and write of the data memory, as well as the selection signals of the multiplexers).

We will now go over all the processes done by the architecture.

Fetching the instruction

As can be seen in Fig 2.2, the execution of the application starts when the program counter (PC) addresses the instruction memory with the first address of the memory, which is initialized as 0x00400000, in hexadecimal. This address reads from the instruction memory the first instruction of the application, which is outputted and feeds the register file. Meanwhile, the same address is forwarded in the architecture to 2 adders, the first one sums it by the number 4 and has the purpose to increment the address so that at the next clock cycle the program counter will address the following instruction of the memory, going sequentially, while the second one has the purpose of calculating the target address for branch operations and unconditional jumps (instructions *jalr, bne, bge, auipc, jal* according to Table 2.1), which require a jump in the address that stops going sequentially, more on that later. The output of these two adders, as well as the output of the ALU, are fed back as the input of the program counter, with a multiplexer controller by the *branch* control signal which will decide what instruction is going to be taken at the next clock cycle. The first input is selected in case of sequential addressing of the instruction memory, the second input is selected in case of conditional jumps (*bne* and *bge*) or unconditional jumps (*jal*), while the third input is selected in case of the *jalr* operation, which has the calculation of the target address performed by the ALU.

Decoding the instruction

The second step of the process consists of the instruction that has been fetched from the instruction memory and is now forwarded to the immediate generator and the register file, which perform the decoding of the instruction. The register file is composed of 32 x 32 bits registers and is able to read the sequence of 32 bits, considering the description of each format and instruction given in Figure 2.1 and Table 2.1, and output the content of the registers addressed by the 5 bits making up the *rs1* and *rs2* to the data_rs1 and data_rs2 lines. In case of the I format, again considering Table 2.1, which has only rs1, and the U and UJ formats, which don't require neither rs1 nor rs2, two multiplexers preceding the ALU, controlled by the 2 control signals *d1_sel* and *ALUsrc*, will decide if the second input of the ALU will be taken from the second output of the register file or from the immediate generator, in the case of the instructions *addi, lw, srai, slli, sw, auipc* (Table 2.1), or if the first input of the ALU will be taken from the first output of the register file or from the current or next address of the operation for instructions *jalr, auipc* and *jal.* Other than that, the immediate generator will also be forwarded to the second adder of the architecture, which require the immediate and the PC value to calculate the target address of the conditional and unconditional jumps, as described earlier.

Executing the instruction

After the decoding and the selection of the input signals to feed to the ALU, that we already discussed, the next process is performed by the ALU, which implements several operations, both arithmetical and logical, and is controlled by the ALU control (Figure 2.3), which selects the proper operation in the current cycle. In particular, the ALU performs the sum for the *add, addi, lw, jalr, sw, auipc* operations (Table 2.1), the subtraction and the xor for the *sub* and *xor* operations respectively, and the shift right and shift left for the *srai* and *slli* operations. It will also tests the condition for the branch operations *bne* and *bge.*

Writing the instruction in the memory

Next, the result of the ALU is forwarded to the data memory, which is addressed at every clock cycle if the control signal memory read is high, and in the case the current instruction doesn't require to access the data memory the output of this stage will be taken directly from the output of the ALU by means of a multiplexer put at the output of the data memory, controlled by a control signal. The second input of the data memory is for writing the data in the memory, which will be used in case of the store operation (*sw*), setting the control signal memory write to a high value.

Writing back of the result

Finally, the last step of the process will be the write back of the result in the destination register of the register file. In this case, as already said, a multiplexer will decide, based on the current instruction, if the result to write back to the register file has to be taken from the data memory or from the output of the ALU.

2.2.2 Pipelined RISC-V

The architecture has been later augmented with the introduction of pipelining registers, to increase the throughput of the design. The RISC-V architecture consists of 5 stages of pipeline, these are the instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB), and corresponds exactly to the division which was given before in order to explain the non pipelined version. The introduction of pipeling registers results in the architecture presented in the following figure:

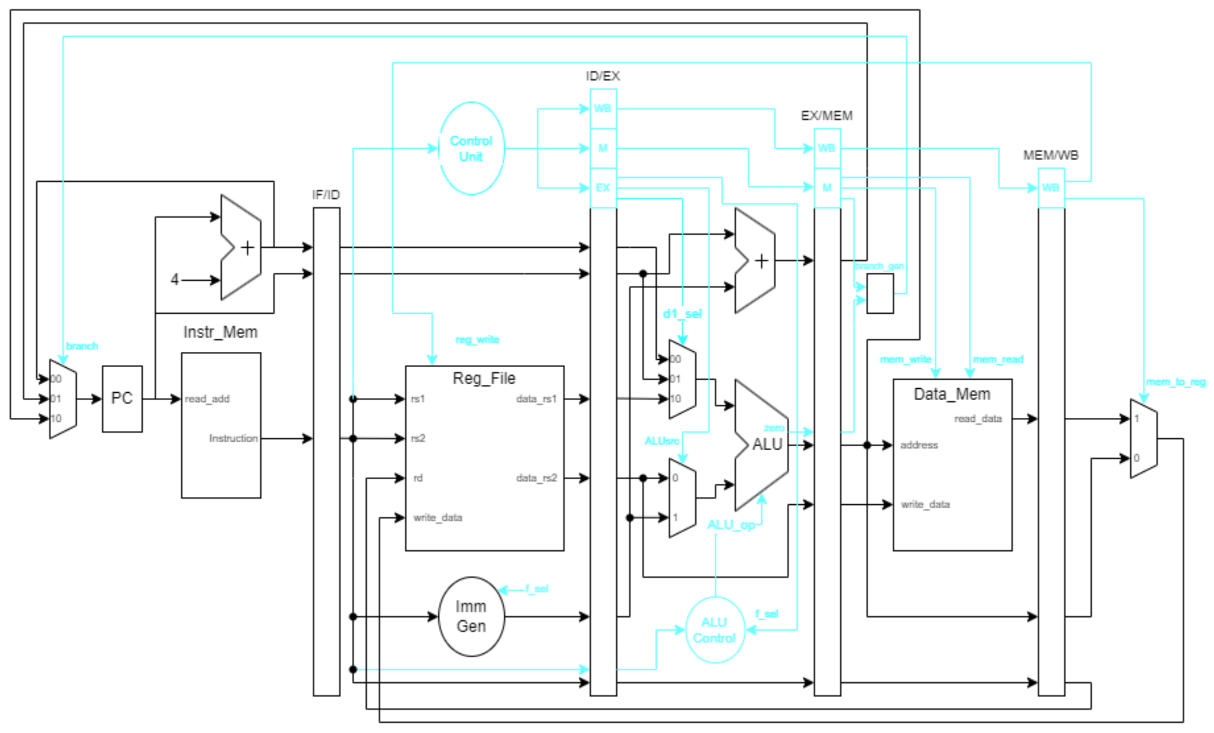


Figure 2.4: Pipelined architecture of the RISC-V

As can be seen, also the control signals have to be forwarded through the pipes, so that the they travel alongside the instruction in the pipes, making all the units perform the correct operation based on the current instruction. This architecture cannot function properly, the reason being that the introduction of pipelining registers generates problems called hazards, which are situations that prevent starting the next instruction in the next clock cycle. The hazards can be of 3 types:

- structural hazards: these happen when a required resource is busy and cannot satisfy the request of an operation;
- data hazards: these happen when there is the need to wait for previous instructions to complete their data read/write in order to perform the following instruction;
- control hazards: these happen in the case of branch operations, since in the pipelined architecture the fetching of the next instruction depends on branch outcome, which result is ready 2 clock cycles after the fetching of the next instruction.

Structural hazards are non existent in this architecture, since they could only be possible if the instruction and data memory were the same memory, that is not the case of the Harvard architecture. Data hazards and control hazards, instead, are a problem that needs to be addressed. Some additional hardware and wiring is required in order to solve these types of hazards, which we will talk about now.

Resolving Data Hazards

Two blocks are needed to resolve data hazards, the first one is a forwarding unit block, the second is an hazard detection unit. The forwarding unit is placed in the EX stage and has the purpose of forwarding at the input of the ALU the data coming from different parts of the architecture, in particular the condition where we need the forwarding arises when an operation that is being processed in the EX stage requires as source operand the result of a previous operation that has not yet made available it's result in the register file, so has not completed the write back (WB) stage yet. This happens if the the source register of the current instruction is equal to the destination register of the instruction that was executed one or two clock cycles before. The forwarding unit then checks the condition if the destination register of the previous one or two operations (MEM/rd or WB/rd) is equal to either the first or second source operand (EX/rs1 and EX/rs2) of the current operation, and based on this condition decides to forward two data to the input of the ALU by driving the selection signal of two multiplexers. The two data forwarded are the output of the ALU taken after the EX/MEM register if MEM/rd = EX/rs1 or EX/rs2, or the write back result after the last multiplexer if WB/rd = EX/rs1 or EX/rs2. The hazard detection unit is placed in the ID stage and has the purpose of checking if the condition where one source operand (ID/rs1 or ID/rs2) is equal to the destination operand of the previous instruction (EX/rd) is a load operation, if that is the case it outputs two control signals, one that goes to the program counter and the other to the first register pipe (IF/ID): these two signals solve the same purpose which is to signal the PC and register pipe to stall for one clock cycle (to not update their output). This is because in the case of data hazards revolving load operations, after the load instruction is detected, we need to wait for one more clock cycle before the forwarding, because the result of the load operation will be available at the end of the MEM stage, and not of the EX stage. Even in this case, the forwarding will be executed by the forwarding unit, so the hazard detection unit purpose will only be to introduce the stall cycle.

Resolving Control Hazards

Since in the flow of instructions there are cases in which **control hazards** happen, a solution to overcome them have been studied. The easiest way to face the problem is by implementing a control for a **flush signal** that is generated by the control unit and it propagates until it reaches the **IF/ID** pipeline register.

In particular, a **comparator** has been placed after the Register file. The comparator's inputs are the data outputs of the Register File. These blocks check if the 2 data are different and if the first one is greater than the second one. This because if there is an instruction in which a branch must be evaluated, in the case of our set of instructions, this can happen only if there is a **bne (branch if it is not equal)** or a **bge (branch if it is greater than)**. The comparator has been placed immediately after the register file in order to don't waste clock cycles in the branch evaluation phase.

So the comparator generates 2 signals: **not_equal** and **greater**. These signals go to the control unit and their assertion in evaluated when the control unit detects the correct opcode and function 3 value of the instruction that may give a possible branch.

Going into details, the **flush signal** is generated when the control unit detects a **jal** or a **jalr** instruction. In these 2 cases the flush is always generated while when it detects the opcode of the **bge** or **bne** instructions it always checks the assertion of the previous quoted signals **greater** and **not_equal** before setting high the flush signal.

Then as said previously, the flush signal reaches the first pipeline register and then resets it to all zeros instruction. The effects on the pipeline is like of having a bubble that propagates at every clock cycle.

2.2.3 Final Design

With all these considerations, as well as implementations that arose when debugging the architecture, the final schematic of the design can be seen in the following figure:

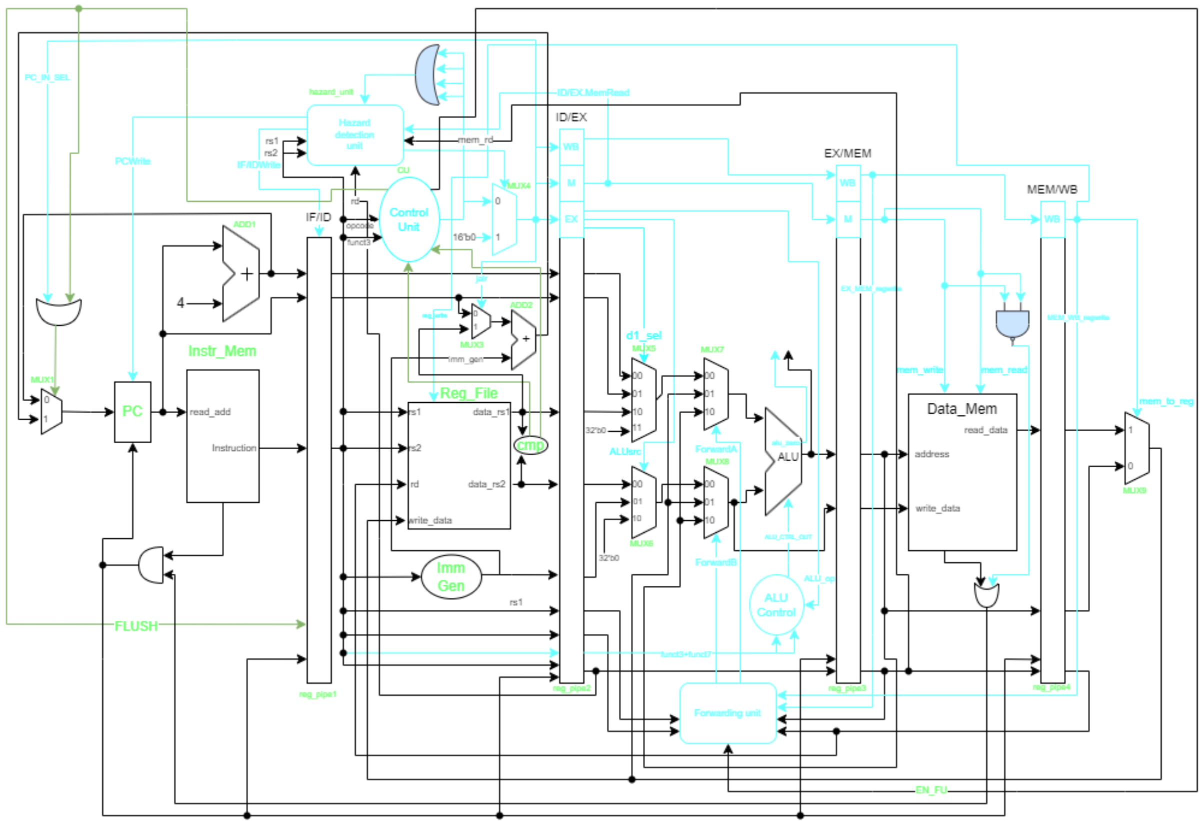


Figure 2.5: Schematic of the implemented RISC-V pipelined architecture

There are several changes that have been implemented to obtain the final structure.
First of all, since both the instruction memory and data memory are sequential elements which require a certain amount of time to perform the reading and writing operations, a system of request/acknowledge has been implemented: according to it, when the input address of the memory changes a request is made, then the SRAM performs it's reading operation and notifies the system when the operation is complete and the data can be read from its output by an acknowledge signal. The request/acknowledge system means that this architecture can work with any memory type, since no assumption has been made considering the delay introduced by the data structures. Due to the delay introduced by the memory elements, the pipe has to be shifted not at every clock cycle, but every several clock cycles, considering the clock cycles it takes to the SRAM to give the acknowledge signal having received the request. For this reason, as can be seen in Figure 2.5, a signal consisting of the AND operation between the two acknowledge signals coming from the memories has been connected to the program counter and four pipe register: in this way the program counter will update it's address and the instruction will travel in the pipe whenever both the instruction memory and data memory have completed their operations. The additional logic regarding the data memory, as can be seen from the figure, has the purpose of making the pipe shift even if the data memory is not being used in the current clock cycle, while as for the instruction memory, there is no need of this additional logic since it is accessed at every clock cycle to read the new instruction.

Unfortunately what this request/acknowledge system also means is that the CPI (cycles per instruction) of this architecture will be for sure larger than 1: in general a pipelined architecture consisting of only one data path can have at best a CPI equal to 1 (one instruction executed in each clock cycle) which is in real case scenarios slightly higher than 1 due to stalls introduced for resolving hazards. In this case, since the memory elements take more than one clock cycle, the achieved CPI will be mainly determined by the behavior of the SRAM. In our particular case, as can be later seen from the simulation of the netlist and the resulting waveforms, the SRAM take 3 clock cycles to perform the reading operation, this sets the CPI of this architecture to a number higher than 3.

Other than this, several minor changes have been made, such as control signals for the forwarding and hazard detection unit, among other things. In particular, the *zero_to_CU* and the connection between the output of the ALU and the multiplexer before the program counter is not required anymore (it has been left floating in Figure 2.5), since the comparison and calculation of the target address for branch operations is now performed in the ID stage.

2.2.4 Encryption

In this section, the task is to modify the previous architecture in such a way that it can support a sort of **decryption unit** for the instructions. As a matter of facts, now a new **encrypted version** of the **minv** application has been created by mean of a **crypt.c** source file, in order to encrypt the instructions that will be contained in the instructions memory.

The logic behind the encryption is that the source file, starting with the group number used as the seed of a pseudo-random generator, generates randomly a sequence of **keys** to encrypt the message in file **main.bin**. In particular, each key encrypts 4 consecutive original instructions just by making the bit-wise exor operation between the instruction and the corresponding key (that of course is also on 4 bytes).

From the RTL point of view, the set of encrypted instructions will substitute the previous instructions memory in a new memory named **I-MEM** while the set of keys will be placed in memory **K-MEM**. The 2 memories have the same dimensions and use the same SSRAM macro. In this section has been treated the block that is in charge of decrypting the instructions in I-MEM and provide their original version to the rest of the circuit. This circuit is described in figure 2.6.

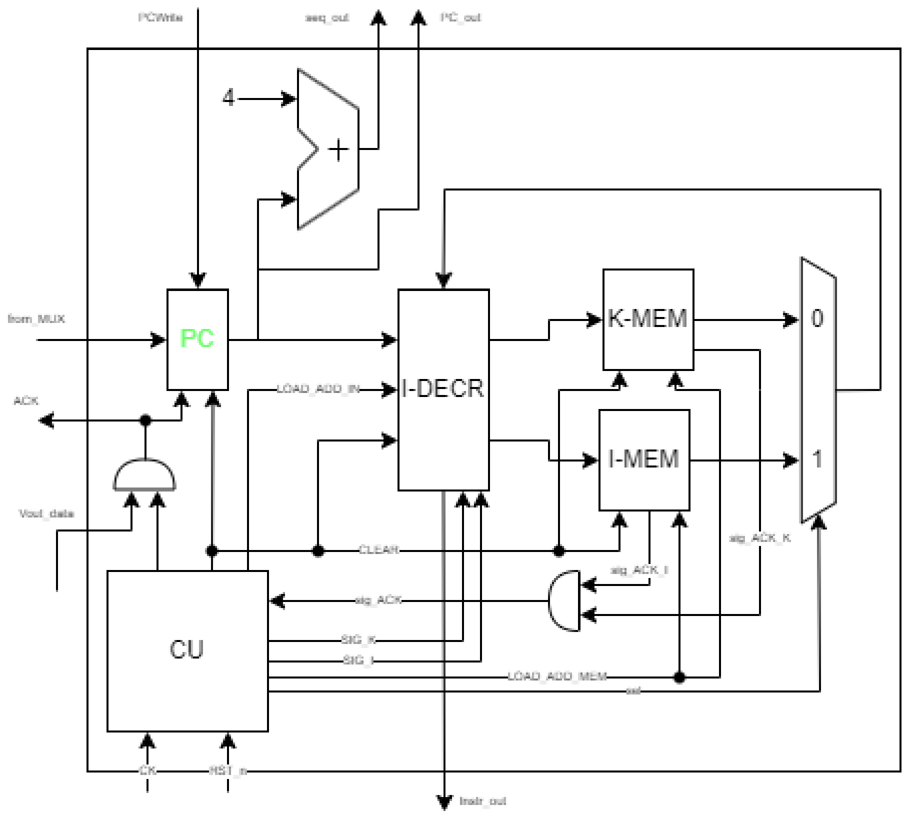


Figure 2.6: Decryption block

The main component of this block is the **I-DECR** one, which is in charge of taking an address from the **program counter** (that now have been moved from the previous circuit inside the decryption block) and starting from it gives at its turn a proper address to both the memories. The multiplexer selects firstly the key from the K-MEM which will be stored in the I-DECR that then will wait for the multiplexer to select the encrypted instruction. After that, when both the key and the encrypted instruction are stored, the I-DECR performs the exor operation to recover the original instruction and send it out from the decryption block, so that it will be managed by the rest of the circuit as done previously.

A particular mention must be done for the **control unit** that manages all the transfers of data in the block. In particular it has to take into account the time needed by the memories to read a data and the time needed by the I-DECR to store both the key and the encrypted instruction. Below in figure 3.1 it is reported a **state diagram** that is useful to understand the behaviour of the control unit.

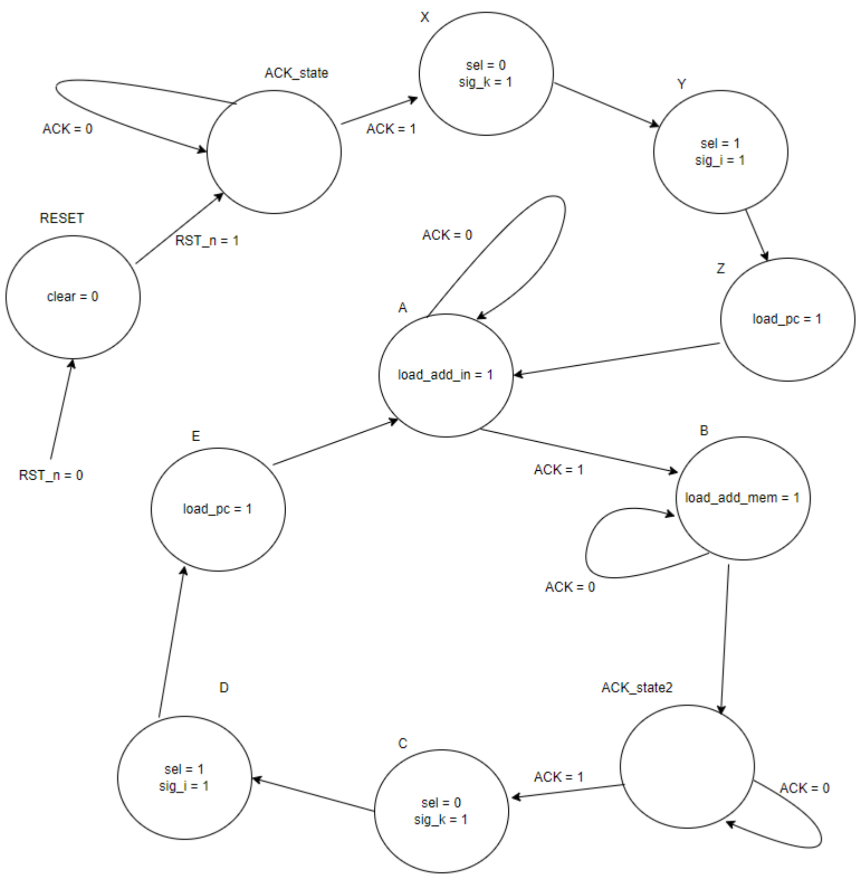


Figure 2.7: State diagram of the control unit

In details, after the **RST_n** (low active signal) goes to 1, the control unit is in a wait state named **ACK_state** where it will waits for the acknowledge signal coming from the memories before going on. During **X, Y and Z** states, the control unit manages the multiplexer in such a way that the data coming from the K-MEM is the first to be stored in the I-DECR block, while the encrypted data of the I-MEM is the second one, then in state Z the control unit makes the PC to take another address and in state **A**, until the acknowledge is low, the I-DECR will load the address from the PC. These states have been inserted to manage the first data that is ready immediately after the reset of the memories and blocks.

After that, when the acknowledge signal becomes high, the control unit goes in state **B** where, until the acknowledge is low, a new address from I-DECR is passed to the memories. It follows another wait state analogous to the first one and it is named **ACK_state2** where the acknowledge signal is waited before proceeding to states **C and D** that make the same job as X and Y. Finally the control unit evolves to state **E** where PC takes a new address and then again to state A in loop.

CHAPTER 3

Simulation and validation of the architecture

In this section, the simulation and the synthesis of the RTL design have been reported and analyzed

3.1 Simulation and synthesis of the design without decryption unit

At first the design has been verified in its original version, without the presence of the decryption unit explained before. In the following figure 3.1 it is reported the wave diagram of the RISC-V:

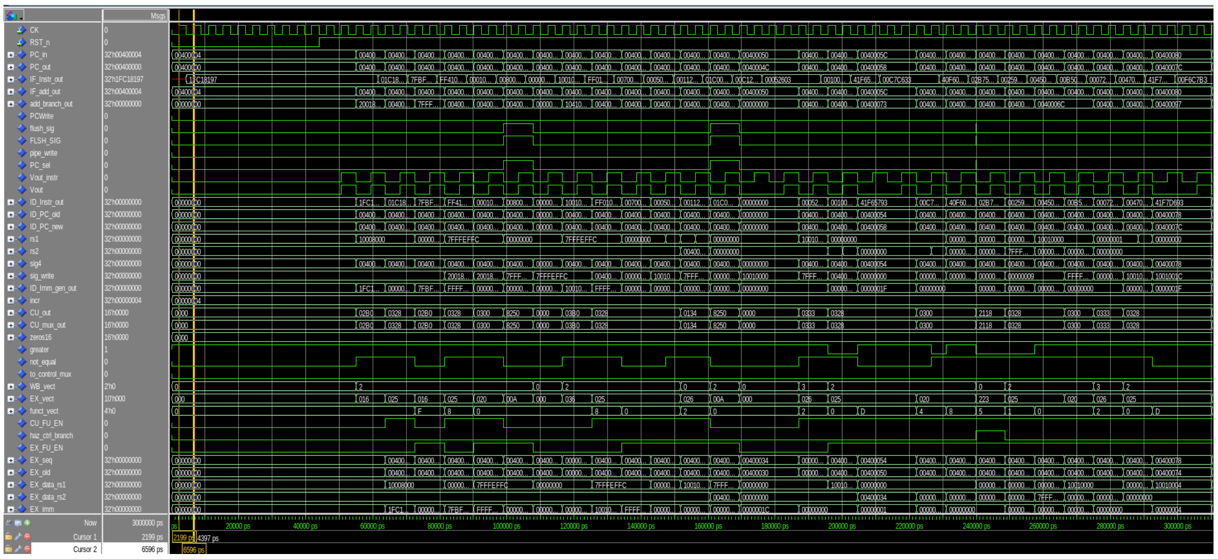
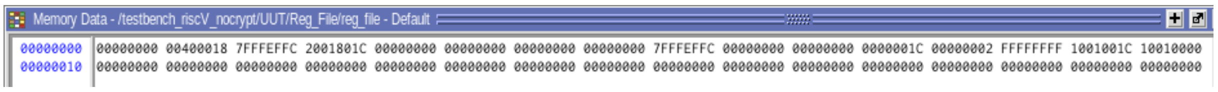


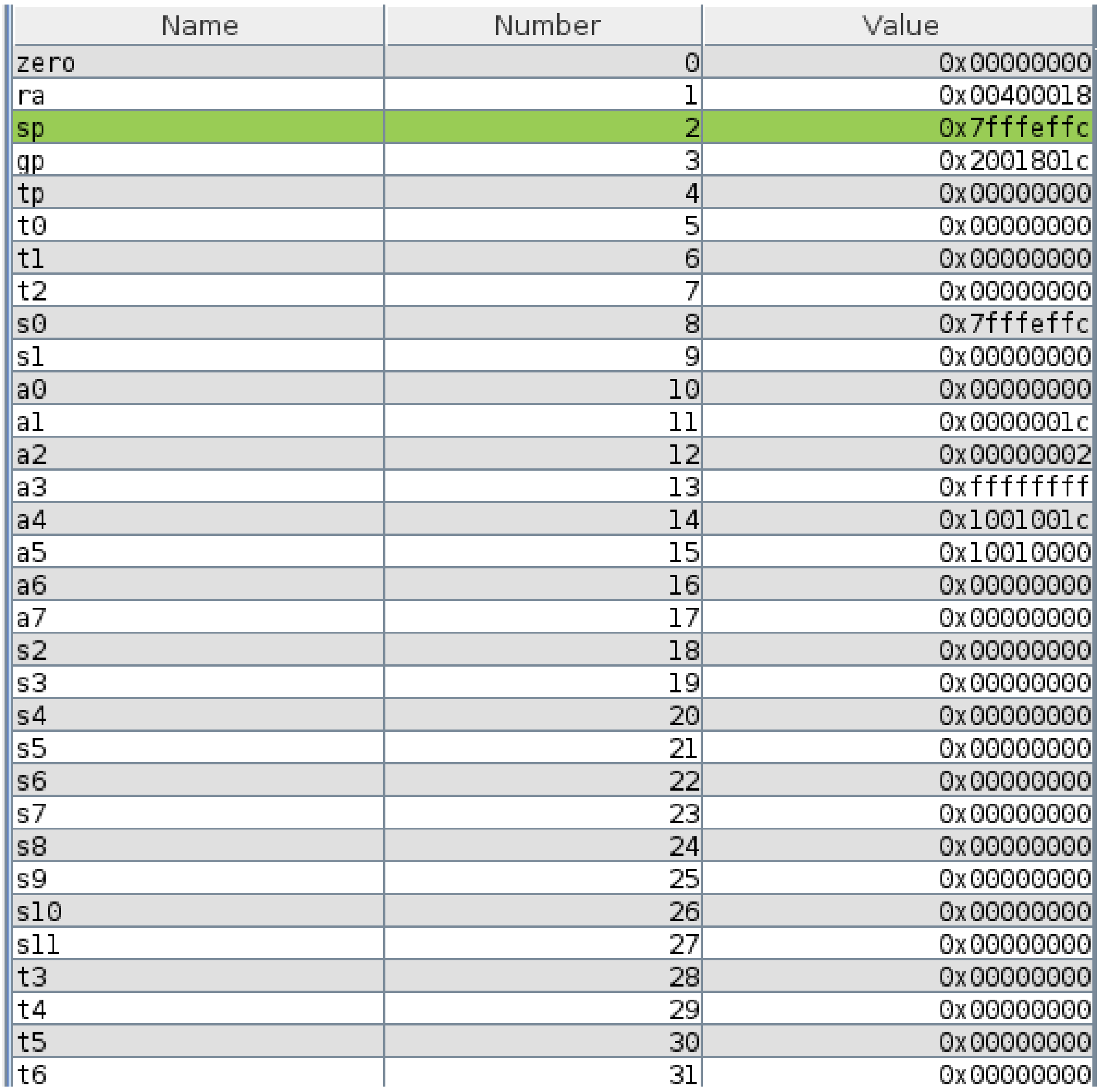
Figure 3.1: Wave diagram of the RISC-V

It is easy to verify that the instruction flow is correct by comparing it to the one of the **RARS instruction simulator** quoted in the first part of the report. In addition, the correct behavior has been verified by looking at the data stored in the register file after the execution of the last instruction. It can be seen that the values stored in the registers are the same of the RARS as reported in figures 3.2 and 3.3:


Memory Data - /testbench_riscV_nocrypt/UUT/Reg_File/reg_file - Default

```
00000000 00000000 00400018 7FFFEFFC 2001801C 00000000 00000000 00000000 00000000 7FFFEFFC 00000000 00000000 0000001C 00000002 FFFFFFFF 1001001C 10010000
00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 3.2: Final values stored in Register File (QUESTASIM)



Name	Number	Value
zero	0	0x00000000
ra	1	0x00400018
sp	2	0x7fffeffc
gp	3	0x2001801c
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x7fffeffc
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x0000001c
a2	12	0x00000002
a3	13	0xffffffff
a4	14	0x1001001c
a5	15	0x10010000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000

Figure 3.3: Final values stored in Register File (RARS)

After that the synthesis of the design has been performed. It must be said that the SSRAM macro has not been included in the script for the Synopsys synthesis but has been added to the **.synopsys_dc.setup** file via the ".db" file. A request for the synthesis was to have all the memory elements defined as "Flip-Flops" and this can be checked by looking at the **elaborate.rep** file in the **syn** folder.

Thanks to the synthesis, the minimum clock period has been found out and it is equal to T_{CK} =

4.4ns. In the following lines are reported the most meaningful results of the timing report:

```
Point                                               Incr      Path
--------------------------------------------------------------------
clock MY_CLK (fall edge)                            2.20      2.20
clock network delay (ideal)                         0.00      2.20
input external delay                                0.00      2.20 f
intr_mem/Rom_mem/clk0 (sram_32_1024_freepdk45)      0.00 #    2.20 f
intr_mem/Rom_mem/dout0[11] (sram_32_1024_freepdk45) 1.83      4.03 r
intr_mem/Instr_out[20] (instructions_rom)           0.00      4.03 r
reg_pipe1/from_IM[20] (IF_ID)                       0.00      4.03 r
reg_pipe1/U245/Z (CLKBUF_X1)                        0.19      4.22 r
reg_pipe1/U227/ZN (AOI22_X1)                        0.03      4.26 f
reg_pipe1/U200/ZN (INV_X1)                          0.03      4.29 r
reg_pipe1/instr_reg[20]/D (DFFR_X1)                 0.01      4.29 r
data arrival time                                             4.29

clock MY_CLK (rise edge)                            4.40      4.40
clock network delay (ideal)                         0.00      4.40
clock uncertainty                                  -0.07      4.33
reg_pipe1/instr_reg[20]/CK (DFFR_X1)                0.00      4.33 r
library setup time                                 -0.03      4.30
data required time                                            4.30
--------------------------------------------------------------------
data required time                                            4.30
data arrival time                                            -4.29
--------------------------------------------------------------------
slack (MET)                                                   0.00
```

This leads to a maximum frequency of about $f_{CK} = 230MHz$. After that, the synthesis has produced a net-list that has been verified by simulation. In figure 3.4 there is the wave diagram:

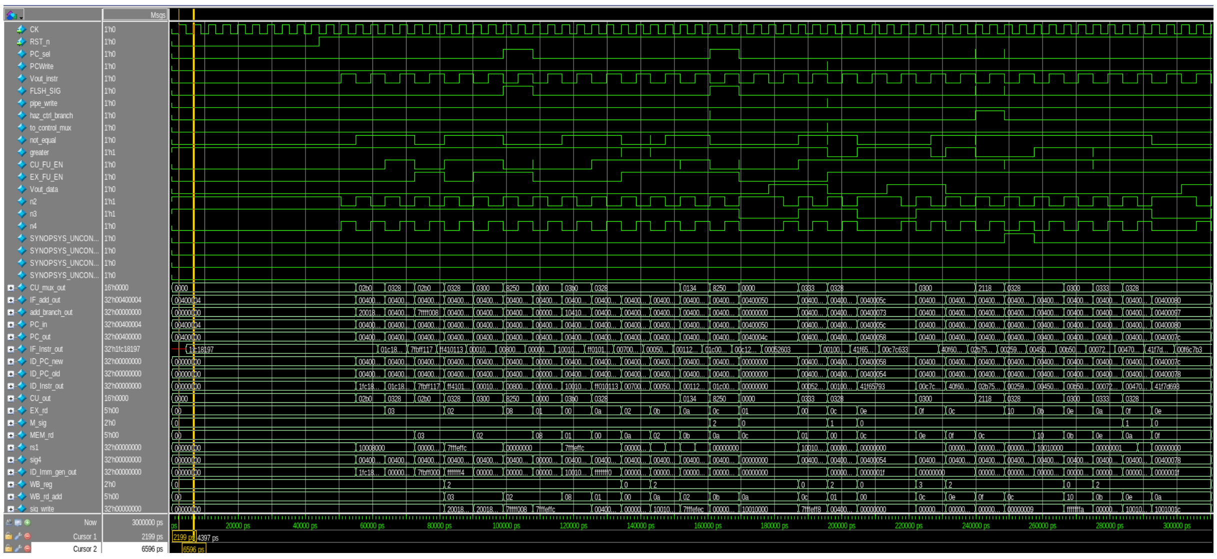


Figure 3.4: Wave diagram of the netlist of the design without decryption unit

As in the previous case, also for the net-list the values of the register file have been analyzed and comparing them with the ones of the RARS file, it can be seen that they are the same.

3.2 Simulation and synthesis of the design with decryption unit

In this section, all the previous steps reported for the design of the RISC-V without decryption unit have been repeated in the case of the inclusion of the decryption block. In the following figure 3.5 is reported the wave diagram obtained with Questasim, showing the proper functioning of the design:

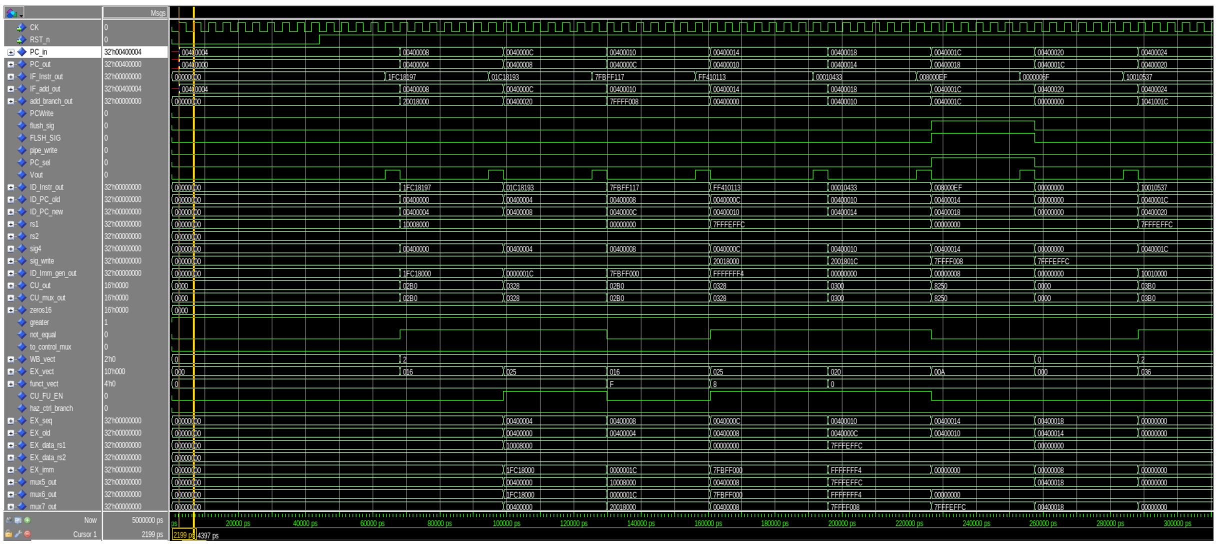
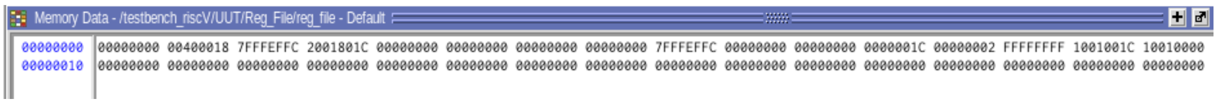


Figure 3.5: Wave diagram of the RISC-V with decryption unit

Also in that case the values in the Register file have been compared with the ones in the RARS simulator. These values obtained by Questasim simulation are reported in figure 3.6:


Memory Data - /testbench_riscV/UUT/Reg_File/reg_file - Default

```
00000000 00000000 00400018 7FFFEFFC 2001801C 00000000 00000000 00000000 00000000 7FFFEFFC 00000000 00000000 0000001C 00000002 FFFFFFFF 1001001C 10010000
00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 3.6: Final values stored in Register File (QUESTASIM)

After that the synthesis has been performed. All the memory elements have been synthesized through Flip-Flops as required, and the same minimum period (so maximum frequency) of the previous case has been obtained: since only the IF stage has been modified, this means that the critical path is not in the IF stage, and so adding more complex logic in this stage has no effect in the clock frequency of the whole system.

As before the most meaningful lines of the timing report are reported:

```
Point                                                    Incr        Path
--------------------------------------------------------------------------
 clock MY_CLK (fall edge)                                2.20        2.20
 clock network delay (ideal)                             0.00        2.20
 input external delay                                    0.00        2.20 f
 data/ram/clk0 (sram_32_1024_freepdk45)                  0.00 #      2.20 f
 data/ram/dout0[31] (sram_32_1024_freepdk45)             1.83        4.03 r
 data/DOUT[0] (data_mem)                                 0.00        4.03 r
 reg_pipe4/data_mem[0] (MEM_WB)                          0.00        4.03 r
 reg_pipe4/U235/Z (CLKBUF_X1)                            0.19        4.22 r
 reg_pipe4/U202/ZN (NAND2_X1)                            0.03        4.25 f
 reg_pipe4/U72/ZN (NAND2_X1)                             0.03        4.28 r
 reg_pipe4/data_mem_out_reg[0]/D (DFFR_X1)               0.01        4.29 r
 data arrival time                                                   4.29

 clock MY_CLK (rise edge)                                4.40        4.40
 clock network delay (ideal)                             0.00        4.40
 clock uncertainty                                      -0.07        4.33
 reg_pipe4/data_mem_out_reg[0]/CK (DFFR_X1)              0.00        4.33 r
 library setup time                                     -0.03        4.30
 data required time                                                  4.30
 -------------------------------------------------------------------------
 data required time                                                  4.30
 data arrival time                                                  -4.29
 -------------------------------------------------------------------------
 slack (MET)                                                         0.01
```

After that, the net-list simulation has been performed. In this case some problems occurred during the simulation of the net-list: as a matter of fact, as can be seen from figure 3.7, the simulation gives undefined results after the reset signal becomes high. Many efforts have been done in order to fix this error, and basically all the files and scripts have been carefully checked but a solution to the problem has not been found in time.

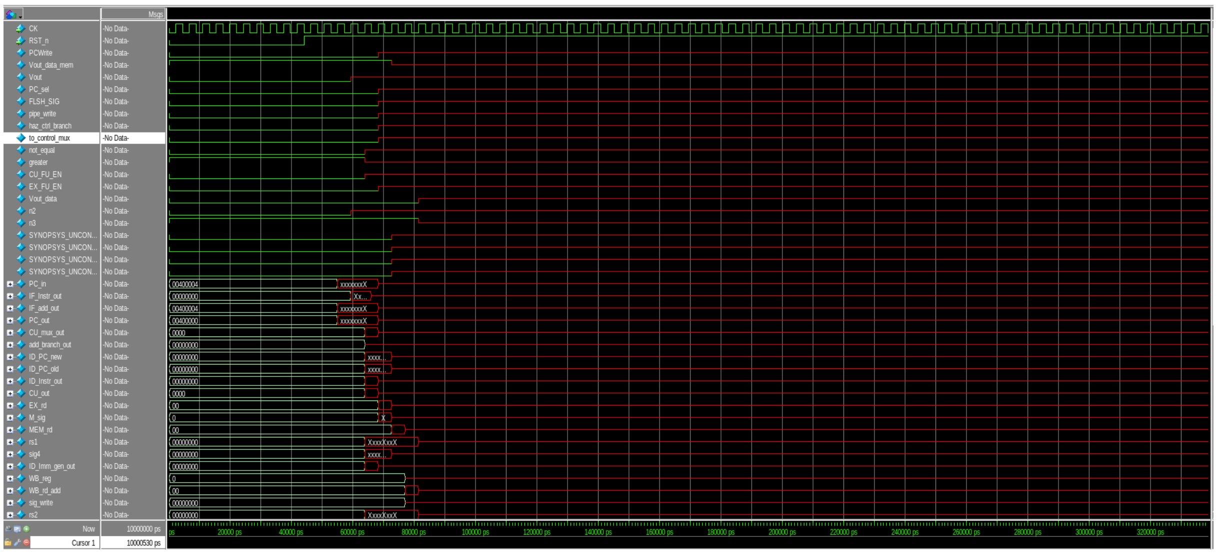


Figure 3.7: Wave diagram of the RISC-V netlist with decryption unit showing an unresolved error